```
#ifndef BST_H
#define BST_H

#include <iostream>
#include <cassert>
using namespace std;

/*--------------------------------
  class BinaryNode
--------------------------------*/
template <class KeyType>
class BinaryNode
{
public:
  BinaryNode<KeyType> *left, *right, *parent;
  KeyType data;

  BinaryNode() : data(KeyType()), left(NULL), right(NULL), parent(NULL) {}
  BinaryNode(const KeyType& val) : data(val), left(NULL), right(NULL), parent(NULL) {}
};

/*--------------------------------
  class BST
--------------------------------*/
template <class KeyType>
class BST
{
public:
  /*---------------Constructor and Destructor----------------*/
  BST(); // default constructor
  BST(const BST<KeyType>& tree); // copy constructor
  ~BST(); // destructor

  /*---------------Public Functions----------------*/
  bool empty() const; // return true if empty; false otherwise
  KeyType* get(const KeyType& k); // return first element with key equal to k
  void insert(KeyType k); // insert k into the tree
  void remove(const KeyType& k); // delete first element with key equal to k
  KeyType* maximum() ; // return the maximum element
  KeyType* minimum() ; // return the minimum element
  KeyType* successor(const KeyType& k); // return the successor of k
  KeyType* predecessor(const KeyType& k); // return the predecessor of k

  /*--------------Public Print Functions---------------*/
  std::string inOrder() const; // return string of elements from an inOrder traversal
  std::string preOrder() const; // return string of elements from a preOrder traversal
  std::string postOrder() const; // return string of elements from a postOrder traversal

public:
  /*---------Class Variables----------*/
  BinaryNode<KeyType> *root;
  int tree_size;

  /*----------Private Functions----------*/
  void insert( KeyType& k, BinaryNode<KeyType> * &ptr, BinaryNode<KeyType> * &dad);
  void remove(const KeyType& k, BinaryNode<KeyType> * &ptr);
  BinaryNode<KeyType> * get2(const KeyType& k, BinaryNode<KeyType> * ptr);
  BinaryNode<KeyType> * maxtree(BinaryNode<KeyType> * ptr);
  BinaryNode<KeyType> * mintree(BinaryNode<KeyType> * ptr);
  BinaryNode<KeyType> * successor(const KeyType& k, BinaryNode<KeyType> * ptr);
  void predecessor(const KeyType& k, BinaryNode<KeyType> * ptr);
  void clearTree(BinaryNode<KeyType> * &ptr);
  void clone(BinaryNode<KeyType> * ptr);
```

```
  void transplant(BinaryNode<KeyType> * u, BinaryNode<KeyType> * v);

  /*----------Private Print Functions----------*/
  std::string inOrder(BinaryNode<KeyType> *ptr) const; // recursively prints the BST in inorde
r format
  std::string preOrder(BinaryNode<KeyType> *ptr) const; // recursively prints the BST in preor
der format
  std::string postOrder(BinaryNode<KeyType> *ptr) const; // recursively prints the BST in post
order format
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const BST<KeyType>& tree);

#include "BST.cpp"

#endif
```

```
#ifndef BST_cpp
#define BST_cpp
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

/*===========================================
BST()     //default constructor
Precondition: None
Postcondition: An empty binary search tree
===========================================*/
template <class KeyType>
BST<KeyType>::BST()
{
  root = NULL;
  tree_size = 0;
}

/*================================================================
BST(const BST<KeyType>& tree);  // copy constructor
Precondition: Must be given a binary tree
Postcondition: Traverses the tree and makes a copy of its values
                to transfer to another tree
================================================================*/
template <class KeyType>
BST<KeyType>::BST(const BST<KeyType>& tree)
{
  root = clone(tree.root);
  tree_size = tree.tree_size;
}

/*======================================================================================
clone(BinaryNode<KeyType> * ptr) // clone method
Precondition: Must be a given binary tree
Postcondition: Basically a recursive preorder traversal over a tree and copy each node
======================================================================================*/
template <class KeyType>
void BST<KeyType>::clone(BinaryNode<KeyType> * ptr)
{
  if (ptr)
  {
    BinaryNode<KeyType> *newNode = new BinaryNode<KeyType>(ptr->data);
    newNode->left = clone(ptr->left);
    newNode->right = clone(ptr->right);
    return newNode;
  }
  return NULL;
}

/*============================================================
 ~BST();                             // destructor
Precondition: None
Postcondition: Deallocates the tree
============================================================*/
template <class KeyType>
BST<KeyType>::~BST()
{
  clearTree(root);
}

/*==========================================================================================
```

```
// clearTree(BinaryNode<KeyType> * ptr) ClearTree Method
Precondition: Must be a given binary tree
Postcondition:  Basically a recursive postorder traversal over a tree and delete each node
==========================================================================================*/
template <class KeyType>
void BST<KeyType>::clearTree(BinaryNode<KeyType> * & ptr)
{
  if (ptr)
  {
    clearTree(ptr->left);
    clearTree(ptr->right);
    delete ptr; // visit => delete node
    ptr = NULL;
  }
}

/*===========================================================================
bool empty() const                       // return whether the MPQ is empty
Precondition: None
Postcondition: Returns true if the binary tree is empty, false otherwise
===========================================================================*/
template <class KeyType>
bool BST<KeyType>::empty() const
{
  return tree_size == 0;
}

/*===================================================================================
get(const KeyType& k)                      // return first element with key equal to k
Precondition: Must be a given binary tree
Postcondition: Recursively calls the hidden 'get' method
===================================================================================*/
template <class KeyType>
KeyType* BST<KeyType>::get(const KeyType& k)
{
  BinaryNode<KeyType> *ptr = get2(k, root);

  if (ptr){
    //cout << ptr->data << " is in tree" << endl;
    return &ptr->data;
  }
  else{
    //cout << k << " is NOT in tree" << endl;
    cout << "Not in tree" <<endl;
    //return (KeyType*) NULL;
  }

}

/*===============================================================================================
==
get(const KeyType& k, BinaryNode>KeyType>* ptr)                      // Protected "get" functi
on
Precondition: Must be a given binary tree
Postcondition: Returns a pointer to a node with key k if one exists; otherwise, returns NIL
=================================================================================================
=*/
template <class KeyType>
BinaryNode<KeyType> * BST<KeyType>::get2(const KeyType& k, BinaryNode<KeyType> * ptr)
{

  while (ptr)
  {
```

```
    if (ptr->data > k)
      ptr = ptr->left;
    else if (ptr->data < k)
      ptr = ptr->right;
    else
      break;
  }
  return ptr;
}

/*===================================================================================
insert(const KeyType& k)             // insert element with key equal k to the tree
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden 'insert' method
===================================================================================*/
template <class KeyType>
void BST<KeyType>::insert(KeyType k)
{
  insert(k, root, root);
}

/*=============================================================================================
================
insert(const KeyType& k, BinaryNode>KeyType>* ptr, BinaryNode<KeyType* dad)     // Protected "
insert" function
Precondition: Must be a given binary tree
Postcondition: Begins at the root of the tree and the pointer ptr traces a simple path downwar
d looking for a
NIL to replace with the input item k
=============================================================================================
==============*/
template <class KeyType>
void BST<KeyType>::insert( KeyType& k, BinaryNode<KeyType> * &ptr, BinaryNode<KeyType> * &dad)
{
  if (ptr == NULL)
  {
    ptr = new BinaryNode<KeyType>(k);
    if(tree_size != 0){
      ptr->parent = dad;
    }
    tree_size++;
  }
  else {
    if (k < ptr->data)
      insert(k, ptr->left, ptr);
    else
      insert(k, ptr->right, ptr);
  }

  if(ptr->parent != NULL){
    //cout << "ptr->parent->data= " << ptr->parent->data <<endl;      DELETE HERE SLATE
  }
}

/*=====================================================================================
remove(const KeyType& k)       // delete first element with key equal k from the tree
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden 'remove' method
=====================================================================================*/
template <class KeyType>
void BST<KeyType>::remove(const KeyType& k)
{
```

```
  remove(k, root);
}

/*==============================================================================================
=============
remove(const KeyType& k, BinaryNode<KeyType> * &ptr)     // Protected "remove" function
Precondition: Must be a given binary tree
Postcondition:
1 - If k has no children, then simply remove it by modifying its parent to replace k with NIL
as its child.
2 - If k has just one child, then we elevate that child to take k’s position in the tree by mo
difying k’s
parent to replace k by k’s child.
3 - If k has 2 children, then we find k’s successor y and have y take k’s position in the tree
. The rest
of k’s original right subtree becomes y’s new right subtree, and k’s left subtree becomes y’s
new left subtree.
================================================================================================
==========*/
template <class KeyType>
void BST<KeyType>::remove(const KeyType& k, BinaryNode<KeyType> * &ptr)
{


  BinaryNode<KeyType> *temp;
  temp = ptr;

  while ((temp->data !=k) and ((temp->left !=NULL) or (temp->right!=NULL))){
    if(temp->data > k){
      temp = temp->left;
    }
    else
      temp = temp->right;
  }



  if (temp->left == NULL)
    transplant(temp, temp->right);
  else if (temp->right == NULL)
    transplant(temp, temp->left);
  else {
    // Has 2 children -- successor(k) must be on the right
//cout << "temp->left->data= " << temp->left->data <<endl;
    BinaryNode<KeyType> *y = mintree(temp->right);
//cout << "y->data="<< y->data <<endl;
    if (y->parent != temp) {
      transplant(y, y->right);
      y->right = ptr->right;
      y->right->parent = y;
    }
    transplant(temp, y);
    y->left = temp->left;
    y->left->parent = y;
  }

  tree_size--;


}

/*==============================================================================================
==
```

```
transplant(BinaryNode<KeyType> * u, BinaryNode<KeyType> * v)      //Transplant Method
Precondition: Must be a given binary tree
Postcondition: Replaces the subtree rooted at node u with the subtree rooted at node v,
u’s parent becomes node v’s parent, and u’s parent ends up having v as its appropriate child.
==============================================================================================
*/
template <class KeyType>
void BST<KeyType>::transplant(BinaryNode<KeyType> * u, BinaryNode<KeyType> * v)
{

  if (u->parent == NULL){
    root = v;
  }
  else if (u == u->parent->left)
    u->parent->left = v;
  else if (u == u->parent->right)
    u->parent->right = v;
  if (v != NULL)
    v->parent = u->parent;

}

/*===========================================================
maximum()             // return the maximum element
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden ’maxtree’ method
============================================================*/
template <class KeyType>
KeyType* BST<KeyType>::maximum()
{
  BinaryNode<KeyType> *temp = maxtree(root);
  assert(temp);
  return &(temp->data);
}

/*================================================================================
maxtree()                         // Protected "maxtree" function
Precondition: Must be a given binary tree
Postcondition: Follows right child pointers from the root until we encounter a NIL
=================================================================================*/
template <class KeyType>
BinaryNode<KeyType> * BST<KeyType>::maxtree(BinaryNode<KeyType> * ptr)
{

  if (ptr)
  {
    while (ptr->right)
      ptr = ptr->right;
  }
  return ptr;
}

/*===========================================================
minimum()             // return the minimum element
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden ’mintree’ method
============================================================*/
template <class KeyType>
KeyType* BST<KeyType>::minimum()
{
  BinaryNode<KeyType> *temp = mintree(root);
  assert(temp);
  return &(temp->data);
```

```
}

/*===============================================================================
mintree()                           // Protected "mintree" function
Precondition: Must be a given binary tree
Postcondition: Follows left child pointers from the root until we encounter a NIL
================================================================================*/
template <class KeyType>
BinaryNode<KeyType> * BST<KeyType>::mintree(BinaryNode<KeyType> * ptr)
{
  if (ptr)
  {
    while (ptr->left)
      ptr = ptr->left;
  }
  return ptr;
}

/*================================================================================
get(const KeyType& k)                   // return the successor of k
Precondition: Must be a given binary tree
Postcondition: Recursively calls the hidden 'successor' method
================================================================================*/
template <class KeyType>
KeyType* BST<KeyType>::successor(const KeyType& k)
{
  BinaryNode<KeyType> *temp;
  temp = successor(k, root);
  return &(temp->data);
}

/*===========================================================================================
==============
successor(const KeyType& k, BinaryNode<KeyType> * ptr)              // Protected "successor" f
unction
Precondition: Must be a given binary tree
Postcondition: If the right subtree of node ptr is non-empty, then the successor of ptr is the
 leftmost node
in ptr's right subtree. On the other hand, if the right subtree of node ptr is empty and ptr h
as a successor
k, then k is the lowest ancestor of ptr whose left child is also an ancestor of ptr
=============================================================================================
============*/
template <class KeyType>
BinaryNode<KeyType> * BST<KeyType>::successor(const KeyType& k, BinaryNode<KeyType> * ptr)
{
  if (ptr == NULL)
    return NULL;
  if (ptr->right) {
    return mintree(ptr->right);
  }
  else {
    BinaryNode<KeyType> *k = ptr->parent;
    while (k != NULL && ptr == k->right) {
      ptr = k;
      k = k->parent;
    }
    return k;
  }
}

/*================================================================================
get(const KeyType& k)                   // return the predecessor of k
```

```
Precondition: Must be a given binary tree
Postcondition: Recursively calls the hidden 'predecessor' method
===========================================================================*/
template <class KeyType>
KeyType* BST<KeyType>::predecessor(const KeyType& k)
{
  predecessor(k, root);
}

/*=============================================================================================
==============
predecessor(const KeyType& k, BinaryNode<KeyType> * ptr)              // Protected "predecesso
r" function
Precondition: Must be a given binary tree
Postcondition: If the left subtree of node ptr is non-empty, then the predecessor of ptr is th
e rightmost node
in ptr's left subtree. On the other hand, if the left subtree of node ptr is empty and ptr has
 a predecessor
k, then k is the lowest ancestor of ptr whose right child is also an ancestor of ptr
===============================================================================================
============*/
template <class KeyType>
void BST<KeyType>::predecessor(const KeyType& k, BinaryNode<KeyType> * ptr)
{
  if (ptr == NULL)
    return NULL;
  if (ptr->left) {
    return maxtree(ptr->left);
  } else {
    BinaryNode<KeyType> *k = ptr->parent;
    while (k != NULL && ptr == k->left) {
      ptr = k;
      k = k->parent;
    }
    return k;
  }
}

/*======================================================================
inOrder()             // return string of elements from an inOrder traversal
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden 'inOrder' method
======================================================================*/
template <class KeyType>
std::string BST<KeyType>::inOrder() const
{
  inOrder(root);
}

/*==========================================================================================
inOrder(BinaryNode<KeyType> *ptr) const                    // Protected "inOrder" function
Precondition: Must be a given binary tree
Postcondition: Prints the key of the root of a subtree between printing the values in its
left subtree and printng those in its right subtree
=========================================================================================*/
template <class KeyType>
std::string BST<KeyType>::inOrder(BinaryNode<KeyType> *ptr) const
{

  if (ptr == NULL)
    return "";

  ostringstream ss;
```

```
    ss << inOrder(ptr->left);
    ss << ptr->data << " ";
    ss << inOrder(ptr->right);
    cout << ss.str() << endl;

  return ss.str();


}

/*================================================================================
preOrder()             // return string of elements from a preOrder traversal
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden 'preOrder' method
================================================================================*/
template <class KeyType>
std::string BST<KeyType>::preOrder() const
{
  preOrder(root);
}

/*==============================================================================================
preOrder(BinaryNode<KeyType> *ptr) const                    // Protected "preOrder" function
Precondition: Must be a given binary tree
Postcondition: Prints the key of the root of a subtree before printing the values in its
left subtree and printng those in its right subtree
==============================================================================================*/
template <class KeyType>
std::string BST<KeyType>::preOrder(BinaryNode<KeyType> *ptr) const
{
  if (ptr == NULL)
    return "";
  ostringstream ss;
  ss << ptr->data << " ";
  ss << preOrder(ptr->left);
  ss << preOrder(ptr->right);
  return ss.str();
}

/*==================================================================================
postOrder()             // return string of elements from a postOrder traversal
Precondition: Must be a given binary tree
Postcondition: Rercursively calls the hidden 'postOrder' method
==================================================================================*/
template <class KeyType>
std::string BST<KeyType>::postOrder() const
{
  postOrder(root);
}

/*==============================================================================================
postOrder(BinaryNode<KeyType> *ptr) const                    // Protected "postOrder" functio
n
Precondition: Must be a given binary tree
Postcondition: Prints the key of the root of a subtree after printing the values in its
left subtree and printng those in its right subtree
==============================================================================================*/
template <class KeyType>
std::string BST<KeyType>::postOrder(BinaryNode<KeyType> *ptr) const
{
  if (ptr == NULL)
    return "";
  ostringstream ss;
```

```
  ss << postOrder(ptr->left);
  ss << postOrder(ptr->right);
  ss << ptr->data << " ";
  return ss.str();
}

#endif
```

```
// test_bst.cpp

#include <iostream>
#include <cassert>
#include "BST.h"

using namespace std;


void test_insert()
{
        BST<int> bst;
        int a=3;
        int b=5;
        int c=2;
        //cout << bst.empty() << endl;
        bst.insert(a);
        //cout << bst.empty() << endl;
        //bst.insert(b);
        bst.insert(c);
        cout << *bst.get(3) << endl;
        bst.remove(3);
        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
        bst.remove(2);
        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
        bst.insert(b);
        cout << *bst.get(5) << endl;
}


void test_remove()
{
        BST<int> bst;
        int a=3;
        int b=5;
        int c=2;
        //cout << bst.empty() << endl;

        *bst.get(3);
        bst.insert(a);
        cout << "3 inserted" << endl;
        //cout << bst.empty() << endl;
        //bst.insert(b);
        bst.insert(c);
        cout << "2 inserted" << endl;
        *bst.get(3);

        bst.remove(3);
        cout << "3 removed" << endl;
        *bst.get(3);


        bst.remove(2);
        cout << "2 removed" << endl;
        *bst.get(2);
        bst.insert(b);
        cout << "5 inserted" << endl;
```

```
        *bst.get(5);
        bst.remove(5);
        cout << "5 removed" << endl;
        *bst.get(5);

        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
cout << "=====================" <<endl;
        bst.insert(a);
        bst.insert(b);
        bst.insert(c);
        cout << "3, 5, and 2 inserted" << endl;
        *bst.get(3);
        *bst.get(5);
        *bst.get(2);

        bst.remove(3);
        cout << "3 removed" << endl;
        *bst.get(3);

        bst.remove(5);
        cout << "5 removed" << endl;
        *bst.get(5);

        bst.remove(2);
        cout << "2 removed" << endl;
        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
cout << "=====================" <<endl;
        bst.insert(a);
        bst.insert(b);
        bst.insert(c);
        cout << "3, 5, and 2 inserted" << endl;
        *bst.get(3);
        *bst.get(5);
        *bst.get(2);

        bst.remove(5);
        cout << "5 was removed" << endl;
        //*bst.get(5);
        //*bst.get(3);

        bst.remove(2);
        cout << "2 was removed" << endl;
        *bst.get(2);

        *bst.get(3);

        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;

        bst.remove(3);
        cout << "3 was removed" << endl;
        *bst.get(3);
        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
```

```
        else
                cout << "tree NOT empty" << endl;

cout << "===================" <<endl;
        int d=6;
        int e=4;
        int f=1;
        bst.insert(a);
        bst.insert(b);
        bst.insert(c);
        bst.insert(d);
        bst.insert(e);
        bst.insert(f);

        cout << "Maximum = " << *bst.maximum() << endl;

        cout << "3, 5, 2, 6, 4, 1 inserted" << endl;
        *bst.get(3);
        *bst.get(5);
        *bst.get(2);
        *bst.get(6);
        *bst.get(4);
        *bst.get(1);

        bst.remove(2);
        cout << "2 was removed" << endl;
        *bst.get(2);
        *bst.get(1);

        bst.remove(3);
        cout << "3 was removed" << endl;
        *bst.get(3);
        *bst.get(4);

        bst.remove(5);
        cout << "5 was removed" << endl;
        *bst.get(5);

        bst.remove(6);
        cout << "6 was removed" << endl;
        *bst.get(6);

        bst.remove(1);
        cout << "1 was removed" << endl;
        *bst.get(1);

        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
        *bst.get(4);


}


void test_maximum()
{
        BST<int> bst;
        int a=3;
        int b=5;
        int c=2;
        int d=6;
```

```
        int e=4;
        int f=1;
        bst.insert(a);
        cout << "Maximum = " << *bst.maximum() << endl;
        bst.insert(b);
        cout << "Maximum = " << *bst.maximum() << endl;
        bst.insert(c);
        cout << "Maximum = " << *bst.maximum() << endl;
        bst.insert(d);
        cout << "Maximum = " << *bst.maximum() << endl;
        bst.insert(e);
        cout << "Maximum = " << *bst.maximum() << endl;
        bst.insert(f);
        cout << "Maximum = " << *bst.maximum() << endl;

        cout << "3, 5, 2, 6, 4, 1 inserted" << endl;
        *bst.get(3);
        *bst.get(5);
        *bst.get(2);
        *bst.get(6);
        *bst.get(4);
        *bst.get(1);

        bst.remove(2);
        cout << "2 was removed" << endl;
        *bst.get(2);
        *bst.get(1);
        cout << "Maximum = " << *bst.maximum() << endl;

        bst.remove(3);
        cout << "3 was removed" << endl;
        *bst.get(3);
        *bst.get(4);
        cout << "Maximum = " << *bst.maximum() << endl;

        bst.remove(5);
        cout << "5 was removed" << endl;
        *bst.get(5);
        cout << "Maximum = " << *bst.maximum() << endl;

        bst.remove(6);
        cout << "6 was removed" << endl;
        *bst.get(6);
        cout << "Maximum = " << *bst.maximum() << endl;

        bst.remove(1);
        cout << "1 was removed" << endl;
        *bst.get(1);
        cout << "Maximum = " << *bst.maximum() << endl;

        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
        *bst.get(4);

}

void test_minimum()
{
        BST<int> bst;
        int a=3;
        int b=5;
```

```
        int c=2;
        int d=6;
        int e=4;
        int f=1;
        bst.insert(a);
        cout << "Minimum = " << *bst.minimum() << endl;
        bst.insert(b);
        cout << "Minimum = " << *bst.minimum() << endl;
        bst.insert(c);
        cout << "Minimum = " << *bst.minimum() << endl;
        bst.insert(d);
        cout << "Minimum = " << *bst.minimum() << endl;
        bst.insert(e);
        cout << "Minimum = " << *bst.minimum() << endl;
        bst.insert(f);
        cout << "Minimum = " << *bst.minimum() << endl;

        cout << "3, 5, 2, 6, 4, 1 inserted" << endl;
        *bst.get(3);
        *bst.get(5);
        *bst.get(2);
        *bst.get(6);
        *bst.get(4);
        *bst.get(1);

        bst.remove(2);
        cout << "2 was removed" << endl;
        *bst.get(2);
        *bst.get(1);
        cout << "Minimum = " << *bst.minimum() << endl;

        bst.remove(3);
        cout << "3 was removed" << endl;
        *bst.get(3);
        *bst.get(4);
        cout << "Minimum = " << *bst.minimum() << endl;

        bst.remove(5);
        cout << "5 was removed" << endl;
        *bst.get(5);
        cout << "Minimum = " << *bst.minimum() << endl;

        bst.remove(6);
        cout << "6 was removed" << endl;
        *bst.get(6);
        cout << "Minimum = " << *bst.minimum() << endl;

        bst.remove(1);
        cout << "1 was removed" << endl;
        *bst.get(1);
        cout << "Minimum = " << *bst.minimum() << endl;

        if(bst.empty() == 1)
                cout << "tree is empty" << endl;
        else
                cout << "tree NOT empty" << endl;
        *bst.get(4);
}

void test_successor()
{
        BST<int> bst;
        int a=3;
```

```
        int b=5;
        int c=2;
        int d=6;
        int e=4;
        int f=1;
        bst.insert(a);
        bst.insert(b);
        bst.insert(c);
        bst.insert(d);
        bst.insert(e);
        bst.insert(f);
        cout << "3, 5, 2, 6, 4, 1 inserted" << endl;

        //bst.inOrder();



        *bst.get(3);
        *bst.get(5);
        *bst.get(2);
        *bst.get(6);
        *bst.get(4);
        *bst.get(1);


        cout <<  *bst.successor(3) << endl;

}

int main()
{
        //test_insert();
        test_remove();
        //test_maximum();
        //test_minimum();
        //test_successor();

        return 0;
}
```

```
#ifndef dictionary_cpp
#define dictionary_cpp
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

/*=====================================================
Dictionary()            // default constructor
Precondition: None
Postcondition: An empty dictionary
=====================================================*/
template <class KeyType>
Dictionary<KeyType>::Dictionary() : BST<KeyType>() {}

/*=======================================================================
Dictionary(const Dictionary<KeyType>& dict)        // copy constructor
Precondition: Must be given a dictionary
Postcondition: Traverses the dictionary and makes a copy of its values
                to transfer to another dictionary
=======================================================================*/
template <class KeyType>
Dictionary<KeyType>::Dictionary(const Dictionary<KeyType>& dict) : BST<KeyType> (dict) {}


/*==========================================================================
bool dict_empty() const            // return whether the dictionary is empty
Precondition: None
Postcondition: Returns true if the dictionary is empty, false otherwise
==========================================================================*/
template <class KeyType>
bool Dictionary<KeyType>::dict_empty() const
{
  return tree_size == 0;
}

/*==================================================================================
dict_get(const KeyType& k)              // return first element with key equal to k
Precondition: Must be a given dictionary
Postcondition: Calls the "get" method inherited from BST class
==================================================================================*/
template <class KeyType>
KeyType* Dictionary<KeyType>::dict_get(const KeyType& k)
{
  get(k);
}

/*==========================================================================================
dict_insert(const KeyType& k)          // insert element with key equal k to the dictionary
Precondition: Must be a given dictionary
Postcondition: Calls the "insert" method inherited from BST class
==========================================================================================*/
template <class KeyType>
void Dictionary<KeyType>::dict_insert(const KeyType& k)
{
  insert(k);
}

/*=============================================================================================
====
dict_remove(const KeyType& k)         // delete first element with key equal k from the dictio
nary
```

```
Precondition: Must be a given dictionary
Postcondition: Calls the "remove" method inherited from BST class
==============================================================================================
==*/
template <class KeyType>
void Dictionary<KeyType>::dict_remove(const KeyType& k)
{
  remove(k);
}

#endif
```

```
#ifndef DICTIONARY_H
#define DICTIONARY_H

#include <iostream>
#include "BST.h"


template <class KeyType>
class Dictionary : public BST<KeyType>
{
public:
  Dictionary(); // default constructor
  Dictionary(const Dictionary<KeyType>& dict); // copy constructor

  bool dict_empty() const; // return whether the dictionary is empty
  KeyType* dict_get(const KeyType& k); // return an element with key k in dictionary
  void dict_insert(const KeyType& k); // insert a new element with key k into the dictionary
  void dict_remove(const KeyType& k); // delete the element with key k from the dictionary

  // Specify that dictionary will be referring to the following members of BST<KeyType>
  using BST<KeyType>::root;
  using BST<KeyType>::tree_size;
  using BST<KeyType>::get;
  using BST<KeyType>::insert;
  using BST<KeyType>::remove;
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const Dictionary<KeyType>& dict);

#include "dictionary.cpp"

#endif
```

```
#include <iostream>
#include <cassert>
#include "dictionary.h"

using namespace std;

void test_insert()
{
  Dictionary<int> dict;
  int a = 10;
  int b = 15;
  int c = 7;

  // cout << dict.dict_empty() << endl;
  dict.dict_insert(a);
  // cout << dict.dict_empty() << endl;
  // dict.dict_insert(b);
  dict.dict_insert(c);
  cout << *dict.dict_get(10) << endl;

  dict.dict_remove(10);
  if(dict.dict_empty() == 1)
    cout << "dictionary is empty" << endl;
  else
    cout << "dictionary NOT empty" << endl;

  dict.dict_remove(7);
  if(dict.dict_empty() == 1)
    cout << "dictionary is empty" << endl;
  else
    cout << "dictionary NOT empty" << endl;

  dict.dict_insert(b);
  cout << *dict.dict_get(15) << endl;
}

void test_remove()
{
  Dictionary<int> dict;
  int a = 10;
  int b = 15;
  int c = 7;
}

int main()
{
  test_insert();
  test_remove();

  return 0;
}
```

```
#ifndef MOVIE_H
#define MOVIE_H

#include <iostream>
#include <stdio.h>
#include <string.h>
#include "dictionary.h"


class Movie
{
public:
  string title;
  string cast;

  Movie() {
    title = "";
    cast = "";
  };

  Movie(string title2, string cast2) {
    title = title2;
    cast = cast2;
  }

  //~Movie();

  bool operator != (const Movie& movie2){
    return title != movie2.title;
  }

  bool operator > (const Movie& movie2){
    return title > movie2.title;
  }


  bool operator < (const Movie& movie2){
    return title < movie2.title;
  }

};


#endif
```

```
// test_movie.cpp

#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <vector>
#include "movie.h"
//#include "dictionary.h"

// Kevin Ly & James Le


using namespace std;

void FileToMovie(){

  Dictionary<Movie> d;

  ifstream file ("movies_mpaa.txt");
  if ( !file.is_open() )//  see if file opened
    cout<<"Could not open file\n";
  else {
    char x;
    bool tab = false;
    string line;
    string lineT;
    string lineC;

    while(getline(file, line)){    //Reads file line by line
      Movie mov;                  // initializes mov each iteration
      int tab = line.find("\t");  // Title and Cast separated by title
      lineT = line.substr(1, tab-9);  //Title contains everything between the quotation marks
      lineC = line.substr(tab+1);  //Cast is Right after tab
      mov.title = lineT;
      mov.cast = lineC;
      d.insert(mov);
      //cout << "|" << mov.title << "| ";
      //cout << "|" << mov.cast << "|" << endl;
    }

    cout << "Please type a movie name (capital letters and spelling matters): ";
    char input[100];
    cin.get(input,100);

    Movie InMovie;
    InMovie.title = input;

    if(input == InMovie.title)
      cout << d.get(InMovie)->cast << endl;
    else
      cout << "Movie Not Fount" << endl;


  }
}

int main()
{
  FileToMovie();
}
```